可靠传输协议实现实验报告

一、实验需求

作业题目：实验2：设计可靠传输协议并编程实现

起止日期： 2025-10-15 08:00:00 ~ 2025-12-27 23:59:59

作业满分： 100

作业说明

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：

1. **连接管理**：包括建立连接、关闭连接和异常处理。
2. **差错检测**：使用校验和进行差错检测。
3. **确认重传**：支持流水线方式，采用选择确认。
4. **流量控制**：发送窗口和接收窗口使用相同的固定大小窗口。
5. **拥塞控制**：实现RENO算法。

实验要求

1. 实现单向数据传输，控制信息需要实现双向交互。
2. 给出详细的协议设计说明。
3. 给出详细的实现方法说明。
4. 利用C或C++语言，使用基本的Socket函数进行程序编写，不允许使用CSocket等封装后的类。
5. 在规定的测试环境中，完成给定测试文件的传输，显示传输时间和平均吞吐率，并观察不同发送窗口和接收窗口大小对传输性能的影响，以及不同丢包率对传输性能的影响。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 提交程序源码、可执行文件和实验报告。

二、协议设计说明

2.1 协议概述

本实验设计实现了一个基于UDP的可靠数据传输协议（RDT），称为RDT-Socket。该协议在不可靠的数据报基础上提供了面向连接、有序交付、差错检测和流量控制等功能。

2.2 数据包格式

包头结构体（64字节）

```
struct PacketHeader {
    uint32_t seq_num;           // 序列号 (4字节)
    uint32_t ack_num;           // 确认号 (4字节)
    uint16_t packet_type;       // 包类型 (2字节)
    uint16_t data_length;       // 数据长度 (2字节)
    uint32_t checksum;          // 校验和 (4字节)
    uint32_t file_size;         // 文件大小 (4字节)
    char filename[32];          // 文件名 (32字节)
    uint8_t reserved[6];        // 保留字段 (6字节)
};
```

包类型定义

类型	值	说明
PKT_SYN	0	连接请求
PKT_SYN_ACK	1	连接应答
PKT_ACK	2	确认包
PKT_DATA	3	数据包
PKT_FIN	4	结束连接
PKT_FIN_ACK	5	结束确认

完整包结构

```
+-------------+--------+
| PacketHeader| Data   |
| (64 bytes)  | (960B) |
+-------------+--------+
总大小：1024字节
```

2.3 连接管理（三次握手）

连接建立流程

```
Client                                Server
  |                                     |
  |-------- SYN (seq=0) ------------> |
  |                                     |
  | <------ SYN-ACK (seq=100) ------- |
  |                                     |
  |-------- ACK (seq=0, ack=100) ---> |
  |                                     |
  |-------- 连接建立 ----------------- |
```

连接关闭流程（四次挥手）

```
Sender                                Receiver
  |                                     |
  |----------- FIN (seq=n) ---------->|
  |                                     |
  |<---------- FIN-ACK (ack=n+1) -----|
  |                                     |
  |-------- 关闭 ---------------------- |
```

2.4 差错检测

采用16位反码和校验（Internet Checksum）算法：

```
1. 将待校验数据分解为16位字
2. 将所有16位字相加（二进制补码运算）
3. 处理进位（16位折叠）
4. 将结果取反得到校验和
```

校验和计算包括：

- 包头（除checksum字段外）的校验和
- 数据部分的校验和
- 总校验和 = (header_checksum + data_checksum) & 0xFFFF

2.5 确认重传机制

发送窗口管理

- **窗口大小**：10个包（固定大小）
- **流水线方式**：支持同时发送多个包
- **选择确认**：记录已确认的包序号

重传策略

- **超时重传**：超过500ms未收到ACK则重传
- **ACK处理**：收到新的ACK时更新send_base，滑动发送窗口
- **重复ACK**：用于快速重传（RENO算法的一部分）

接收窗口管理

- **窗口大小**：10 × DATA_SIZE字节（固定大小）
- **乱序缓存**：使用map缓存乱序到达的包
- **顺序交付**：只有接收到seq_num == recv_base的包才能交付给应用层

2.6 流量控制

发送窗口约束：

```
send_window.size() < min(WINDOW_SIZE, effective_window)

其中：
WINDOW_SIZE = 10（固定窗口大小）
effective_window = min(WINDOW_SIZE, cwnd)（受拥塞控制限制）
```

接收窗口约束：

```
recv_base <= seq_num < recv_base + WINDOW_SIZE × DATA_SIZE
```

2.7 拥塞控制（RENO算法）

两个阶段

状态	说明	cwnd增长策略
SLOW_START	慢启动	每个ACK增加1，指数增长
CONGESTION_AVOIDANCE	拥塞避免	约每RTT增加1，线性增长（通过ACK累加实现）

状态转移

```
初始化：cwnd=1, ssthresh=10

SLOW_START:
  - 收到新ACK: cwnd++, 如果cwnd>=ssthresh则转到CONGESTION_AVOIDANCE
  - 3个重复ACK: 触发快速重传与窗口调整（ssthresh=cwnd/2, cwnd=ssthresh+3）
  - 超时: ssthresh=cwnd/2, cwnd=1, 转到SLOW_START

CONGESTION_AVOIDANCE:
  - 收到新ACK: 通过ACK累加（约每RTT+1）更新cwnd
  - 3个重复ACK: 触发快速重传与窗口调整（ssthresh=cwnd/2, cwnd=ssthresh+3）
  - 超时: ssthresh=cwnd/2, cwnd=1, 转到SLOW_START
```

三、实现方法说明

本节按照实验要求的功能点（连接管理、差错检测、确认重传、流量控制、拥塞控制）对代码实现进行说明，并给出关键代码片段。

3.1 整体架构与代码结构

项目采用 C++ 实现，在 Windows 平台使用 Winsock2 的基础 Socket API 完成 UDP 通信（如 `socket/bind/sendto/recvfrom/setsockopt/closesocket`），未使用 MFC 的 `CSocket` 等封装类。

代码分为三层：

1. **协议定义层**：`protocol.h`（包格式、常量、校验和）。
2. **可靠传输层**：`rdt_socket.h/.cpp`（`RdtSocket`，实现连接管理、可靠传输、窗口与拥塞控制）。
3. **应用层**：`sender.cpp` / `receiver.cpp`（参数解析 + 调用 `RdtSocket` 接口完成文件传输）。

应用层调用链：

```
// sender.cpp：bind -> connect -> sendFile -> close
RdtSocket sender;
sender.bind("127.0.0.1", 0);
sender.connect(remote_ip, remote_port);
sender.sendFile(file_path);
sender.close();
```

```
// receiver.cpp：listen -> accept -> recvFile -> close
RdtSocket receiver;
receiver.listen(local_port);
RdtSocket* client = receiver.accept();
client->recvFile(save_path);
client->close();
delete client;
```

3.2 连接管理：建立连接、关闭连接与异常处理

3.2.1 建立连接（三次握手）

在 UDP 之上模拟面向连接：客户端发送 `SYN`，服务端回复 `SYN-ACK`，客户端回复 `ACK`。

发送端 `connect()` 的核心逻辑：

```
Packet syn_pkt;
syn_pkt.header.packet_type = PKT_SYN;
syn_pkt.header.seq_num = local_seq;
syn_pkt.header.data_length = 0;
syn_pkt.header.checksum = 0;
syn_pkt.header.checksum = calculateChecksum(&syn_pkt.header,
    sizeof(syn_pkt.header) - sizeof(syn_pkt.header.checksum));
sendPacket(syn_pkt);

Packet ack_pkt;
while (true) {
    if (recvPacket(ack_pkt, 100) && ack_pkt.header.packet_type == PKT_SYN_ACK) {
        remote_seq = ack_pkt.header.seq_num;
        recv_base = remote_seq;

        Packet final_ack;
        final_ack.header.packet_type = PKT_ACK;
        final_ack.header.seq_num = local_seq;
        final_ack.header.ack_num = remote_seq;
        final_ack.header.checksum = 0;
        final_ack.header.checksum = calculateChecksum(&final_ack.header,
            sizeof(final_ack.header) - sizeof(final_ack.header.checksum));

        if (sendPacket(final_ack)) { connected = true; return true; }
    }
    // 超过 CONNECT_TIMEOUT_MS 则连接失败
}
```

接收端 `accept()` 的核心逻辑：

```
recvfrom(sock, (char*)&syn_pkt, sizeof(syn_pkt), 0, (sockaddr*)&remote_addr,
&addr_len);
if (syn_pkt.header.packet_type != PKT_SYN) return nullptr;

RdtSocket* new_sock = new RdtSocket();
new_sock->sock = this->sock;
new_sock->remote_addr = this->remote_addr;
new_sock->local_addr = this->local_addr;
new_sock->remote_seq = syn_pkt.header.seq_num;
new_sock->recv_base = syn_pkt.header.seq_num;
```

```
new_sock->local_seq = 100;

Packet syn_ack;
syn_ack.header.packet_type = PKT_SYN_ACK;
syn_ack.header.seq_num = new_sock->local_seq;
syn_ack.header.ack_num = new_sock->remote_seq;
syn_ack.header.checksum = 0;
syn_ack.header.checksum = calculateChecksum(&syn_ack.header,
    sizeof(syn_ack.header) - sizeof(syn_ack.header.checksum));
new_sock->sendPacket(syn_ack);

Packet ack_pkt;
if (new_sock->recvPacket(ack_pkt, CONNECT_TIMEOUT_MS) &&
ack_pkt.header.packet_type == PKT_ACK) {
    new_sock->connected = true;
    return new_sock;
}
```

3.2.2 关闭连接（FIN/FIN-ACK）

文件传输结束后由发送端主动发 `FIN`，接收端收到后回 `FIN-ACK`：

```
Packet fin;
fin.header.packet_type = PKT_FIN;
fin.header.seq_num = seq;
fin.header.ack_num = recv_base;
fin.header.checksum = 0;
fin.header.checksum = calculateChecksum(&fin.header,
    sizeof(fin.header) - sizeof(fin.header.checksum));
sendPacket(fin);

Packet fin_ack;
if (recvPacket(fin_ack, CONNECT_TIMEOUT_MS) && fin_ack.header.packet_type ==
PKT_FIN_ACK) {
    log("[SEND] Connection closed");
}
connected = false;
```

```
// recvFile() 中收到 FIN 的处理
Packet fin_ack;
fin_ack.header.packet_type = PKT_FIN_ACK;
fin_ack.header.seq_num = local_seq;
fin_ack.header.ack_num = data_pkt.header.seq_num;
fin_ack.header.checksum = 0;
fin_ack.header.checksum = calculateChecksum(&fin_ack.header,
    sizeof(fin_ack.header) - sizeof(fin_ack.header.checksum));
sendPacket(fin_ack);
connected = false;
```

3.2.3 异常处理（收包超时）

收包超时通过 `SO_RCVTIMEO` 实现，`recvPacket()` 超时返回 false，上层可据此重试/重传：

```
int timeout = timeout_ms;
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout));
int n = recvfrom(sock, (char*)&pkt, sizeof(Packet), 0, (sockaddr*)&remote_addr,
&addr_len);
return (n != SOCKET_ERROR);
```

3.3 差错检测：校验和（Checksum）

校验和函数定义在 `protocol.h`，使用 16 位反码和校验的实现方式（将所有字节相加、折叠进位、取反）：

```
inline uint32_t calculateChecksum(const void* buffer, size_t length) {
    uint32_t sum = 0;
    const uint8_t* data = (const uint8_t*)buffer;
    for (size_t i = 0; i < length; i++) sum += data[i];
    while (sum >> 16) sum = (sum & 0xFFFF) + (sum >> 16);
    return (~sum) & 0xFFFF;
}
```

发送端对 `header(不含checksum字段) + data` 分别计算并合成 checksum：

```
data_pkt.header.checksum = 0;
uint32_t header_checksum =
    calculateChecksum(&data_pkt.header, sizeof(data_pkt.header) -
sizeof(data_pkt.header.checksum));
uint32_t data_checksum = calculateChecksum(data_pkt.data, to_send);
data_pkt.header.checksum = (header_checksum + data_checksum) & 0xFFFF;
```

接收端复算并比对，不一致则丢弃该包（依赖后续重传）：

```
uint32_t received_checksum = data_pkt.header.checksum;
data_pkt.header.checksum = 0;
uint32_t header_checksum =
    calculateChecksum(&data_pkt.header, sizeof(data_pkt.header) -
sizeof(data_pkt.header.checksum));
uint32_t data_checksum = calculateChecksum(data_pkt.data,
data_pkt.header.data_length);
uint32_t expected = (header_checksum + data_checksum) & 0xFFFF;
if (expected != received_checksum) { continue; }
```

3.4 确认重传：流水线发送 + 重传策略

3.4.1 序列号与 ACK 语义

本实现采用“字节序号”作为 `seq_num`：每发送 `to_send` 字节就 `seq += to_send`。接收端回 `ACK(ack_num=recv_base)`，表示累计确认到 `recv_base`（下一段期望接收的字节序号）。

3.4.2 流水线（发送窗口）

发送端用 `send_window` 保存所有“已发送但未确认”的数据包（用于滑窗与重传）：

```
struct SendWindowEntry {
    Packet packet;
    std::chrono::steady_clock::time_point send_time;
    uint32_t retransmit_count;
};
std::map<uint32_t, SendWindowEntry> send_window;
```

流水线技术是现代协议的核心，它允许发送端在等待接收端的 ACK 期间继续发送新的数据包，而不必等待每个包都被确认后才能继续。这大幅提高了网络利用率，特别是在网络延迟较高的场景下。本实现通过维护一个"发送窗口"来实现流水线。发送窗口用 `send_window` 这个 map 数据结构来保存所有"已发送但未确认"的数据包。map 的 key 是数据包的序列号，value 是一个 `SendWindowEntry` 结构体，包含了完整的包内容、发送时间戳和重传次数计数。发送数据时，发送端首先检查是否可以继续发送（通过 `canSendPacket()` 判断窗口是否有空间），如果可以，就立即将数据包加入发送窗口并通过 UDP 发送出去。这个过程中，时间戳和重传次数都被初始化，为后续的超时检测和重传计数做好准备。

```
SendWindowEntry entry;
entry.packet = data_pkt;
entry.send_time = std::chrono::steady_clock::now();
entry.retransmit_count = 0;
send_window[seq] = entry;
sendPacket(data_pkt);
```

发送窗口的大小受到两个限制：一个是固定的流量控制窗口 `WINDOW_SIZE`（通常为 10），另一个是动态的拥塞控制窗口 `cwnd`。发送端取两者的最小值作为有效发送窗口大小，这样既保证了接收端不会被过多数据淹没，也避免了在网络拥塞时继续高速发送。

3.4.3 ACK 处理：滑窗、重复 ACK 与快速重传

当接收端收到完整的数据并写入文件后，它会回复一个 ACK 包。ACK 包中的 `ack_num` 字段表示接收端期望接收的下一个字节的序列号，也就是说，所有 `seq_num < ack_num` 的数据都已经被正确接收了。这被称为"累计确认"。在发送端，对 ACK 的处理分为两种情况：新的 ACK 和重复的 ACK。新 ACK（`ack_seq > last_ack_seq`）到达时，发送端执行"滑动窗口"操作。这个操作的含义是：删除所有 `seq < ack_seq` 的在途包，因为这些包已经被接收端确认了，不再需要保存。同时将 `last_ack_seq` 更新为最新的确认序号。这样做之后，发送窗口就向前移动了，发送端可以继续发送新的数据包。

```
void slideWindow(uint32_t ack_seq) {
    auto it = send_window.begin();
    while (it != send_window.end() && it->first < ack_seq) {
        it = send_window.erase(it);
    }
}
```

重复 ACK（`ack_seq == last_ack_seq`）意味着接收端连续多次收到相同的 ACK，这通常表示某个中间的包已经丢失。当累计到 3 个重复 ACK 时（即同一个序列号被确认了 4 次总计），TCP Reno 采用"快速重传"机制：不等待 500ms 的超时，而是立即重传"最早未确认包"。这个设计基于以下观察：如果接收端连续发送相同的 ACK，说明它在期待某个特定的数据包，而后续数据已经被正确接收。这比等待超时更快、更高效地恢复网络传输。

```
} else if (ack_seq == last_ack_seq) {
    onDuplicateAck();
    if (dup_ack_count == 3) {
        if (!send_window.empty()) {
            auto first_unacked = send_window.begin();
            sendPacket(first_unacked->second.packet);
            first_unacked->second.send_time = std::chrono::steady_clock::now();
            first_unacked->second.retransmit_count++;
        }
    }
}
```

这个快速重传机制的关键优势在于：它能够在**网络状况相对较好**但出现偶发丢包的场景中，迅速恢复，而不必等待完整的 RTO（重传超时）。相比之下，如果某个包丢失且接收端没有后续数据要发送，就不会产生重复 ACK；在这种情况下，只能依靠超时机制来检测丢包。

3.4.4 超时重传

即使实现了快速重传机制，超时重传仍然是必不可少的。快速重传依赖于接收端的反馈（重复 ACK），但如果一个数据包丢失且接收端之后没有任何数据包要发送（即没有"触发"重复 ACK 的机制），那么发送端就无法通过重复 ACK 来检测这个丢失。在这种情况下，只有超时机制能够可靠地检测到丢包。本实现使用 `TIMEOUT_MS=500` 作为重传超时阈值。每当检测到发送窗口内某个包的发送时间距离当前时刻超过 500ms 时，就认为该包已经丢失，需要立即重传。重传时更新该包的发送时间戳，使其重新开始超时计时。同时，超时事件触发了拥塞控制的"退避"逻辑（`onTimeout()`），在这种情况下，网络状况可能更加恶劣，应该更激进地降低拥塞窗口。

```
if (elapsed > TIMEOUT_MS) {
    sendPacket(entry.second.packet);
    onTimeout();
}
```

在 `send_window` 中记录 `retransmit_count` 对诊断和统计也有重要意义。如果某个包被重传了许多次仍然超时，这强烈表明网络状况极差，或者远端主机可能已经离线。本实现可以在将来扩展为：当 `retransmit_count` 超过某个阈值（比如 5 次）时，主动放弃该传输并报错，而不是无限重试。

此外，超时重传与快速重传的区别在于：

- **快速重传**：发生在网络有"轻微丢包"的情况下，接收端仍在反馈，只是某个包丢失了。恢复相对温和，使用快速恢复（cwnd = ssthresh + 3）。
- **超时重传**：发生在网络"严重恶化"或"甚至可能断连"的情况下，接收端长时间没有任何反馈。恢复激进，重新进入慢启动（cwnd = 1）。

> 说明：接收端能够缓存乱序包（选择性接收），但 ACK 字段为累计确认（未实现 TCP SACK 的位图/区间回传）。

3.4.5 选择确认（Selective Acknowledgment, SACK）

选择确认（SACK）是一种优化的确认机制，允许接收端通知发送端其已收到的非连续数据块，从而减少不必要的重传，提高传输效率。传统的累计确认（ACK）只能确认连续收到的数据范围，而 SACK 通过显式地告知发送端哪些数据已经收到，即使这些数据是乱序的，也能避免发送端重复发送这些数据。

在本实现中，选择确认的逻辑分为以下几个步骤：

接收端：生成 SACK 块

接收端通过 `generateSackBlocks` 方法扫描 `recv_buffer`，找出所有不连续的已缓存数据块，并将这些数据块记录为 SACK 块。每个 SACK 块包含两个字段：起始序号（`start`）和结束序号（`end`），表示该数据块的范围。

具体逻辑如下：

- 遍历 `recv_buffer` 中的所有数据包。
- 如果发现数据包之间存在间隙，则将当前数据块记录为一个 SACK 块。
- 最多生成 `MAX_SACK_BLOCKS` 个 SACK 块，避免数据部分超出限制。
- 最后，将所有生成的 SACK 块编码到 ACK 包的数据部分。

以下是生成 SACK 块的核心代码：

```
void RdtSocket::generateSackBlocks(SackBlock* blocks, uint8_t& count) {
    count = 0;
    if (recv_buffer.empty()) return;

    uint32_t prev_end = recv_base;
    SackBlock current_block;
    bool in_block = false;

    for (auto& entry : recv_buffer) {
        uint32_t seq = entry.first;
        if (seq < recv_base) continue;

        if (seq > prev_end) {
            if (in_block && count < MAX_SACK_BLOCKS) {
                blocks[count++] = current_block;
                in_block = false;
            }
        }

        if (!in_block) {
            current_block.start = seq;
            in_block = true;
        }

        const Packet& pkt = entry.second;
        current_block.end = seq + pkt.header.data_length;
        prev_end = current_block.end;
    }

    if (in_block && count < MAX_SACK_BLOCKS) {
        blocks[count++] = current_block;
    }
}
```

接收端：发送带 SACK 的 ACK

接收端在生成 SACK 块后，通过 `sendAckWithSack` 方法将这些块编码到 ACK 包的数据部分，并发送给发送端。发送逻辑如下：

- 调用 `generateSackBlocks` 方法生成 SACK 块。
- 将 SACK 块编码到 ACK 包的 `data` 字段中。
- 计算校验和并发送 ACK 包。

以下是发送带 SACK 的 ACK 的核心代码：

```
bool RdtSocket::sendAckWithSack(uint32_t ack_seq) {
    Packet ack;
    ack.header.packet_type = PKT_ACK;
    ack.header.seq_num = local_seq;
    ack.header.ack_num = ack_seq;

    SackBlock sack_blocks[MAX_SACK_BLOCKS];
    uint8_t sack_count = 0;
    generateSackBlocks(sack_blocks, sack_count);

    uint16_t data_len = 0;
    if (sack_count > 0) {
        data_len = encodeSackBlocks(sack_blocks, sack_count, ack.data,
DATA_SIZE);
    }

    ack.header.data_length = data_len;
    ack.header.checksum = 0;
    ack.header.checksum = calculateChecksum(&ack.header,
                                            sizeof(ack.header) -
sizeof(ack.header.checksum));
    if (data_len > 0) {
        ack.header.checksum += calculateChecksum(ack.data, data_len);
    }

    return sendPacket(ack);
}
```

发送端：处理 SACK 信息

发送端在收到带 SACK 的 ACK 后，会解析其中的 SACK 块，并更新 `sacked_packets` 集合，记录哪些数据包已经被确认。随后，在重传逻辑中，发送端会跳过这些已确认的数据包，仅重传未确认的数据包。

以下是重传逻辑的核心代码：

```
for (auto& entry : send_window) {
    if (sacked_packets.find(entry.first) != sacked_packets.end()) {
        log("[SACK] Packet (seq=%u) is SACKED, skipping retransmit",
entry.first);
        continue;
    }

    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(
        std::chrono::steady_clock::now() - entry.second.send_time).count();
    if (elapsed > TIMEOUT_MS) {
        log("[RETX] Packet timeout, retransmitting (seq=%u)", entry.first);
        entry.second.send_time = std::chrono::steady_clock::now();
        entry.second.retransmit_count++;
        sendPacket(entry.second.packet);
        onTimeout();
    }
}
```

选择确认的优势

选择确认机制显著提升了传输效率，尤其是在高丢包率的网络环境下：

- **减少重传**：发送端仅重传未确认的数据包，避免重复发送已收到的数据。
- **提高吞吐量**：通过减少不必要的重传，发送端可以更快地发送新数据。
- **优化网络利用率**：减少了无效数据的传输，降低了网络负载。

在本实验的实现中，选择确认机制与流水线传输、超时重传等功能相结合，进一步增强了协议的可靠性和性能。

四、测试环境配置和使用说明

4.1 测试环境要求

- **操作系统**：Windows 10/11
- **编译器**：g++ (MinGW)
- **网络模拟工具**：Router.exe

4.2 编译方法

```
cd d:\study\computer_net\l2
g++ -Wall -std=c++11 -I./ -c -o rdt_socket.o rdt_socket.cpp
g++ -Wall -std=c++11 -I./ -o sender.exe sender.cpp rdt_socket.o -lws2_32
g++ -Wall -std=c++11 -I./ -o receiver.exe receiver.cpp rdt_socket.o -lws2_32
```

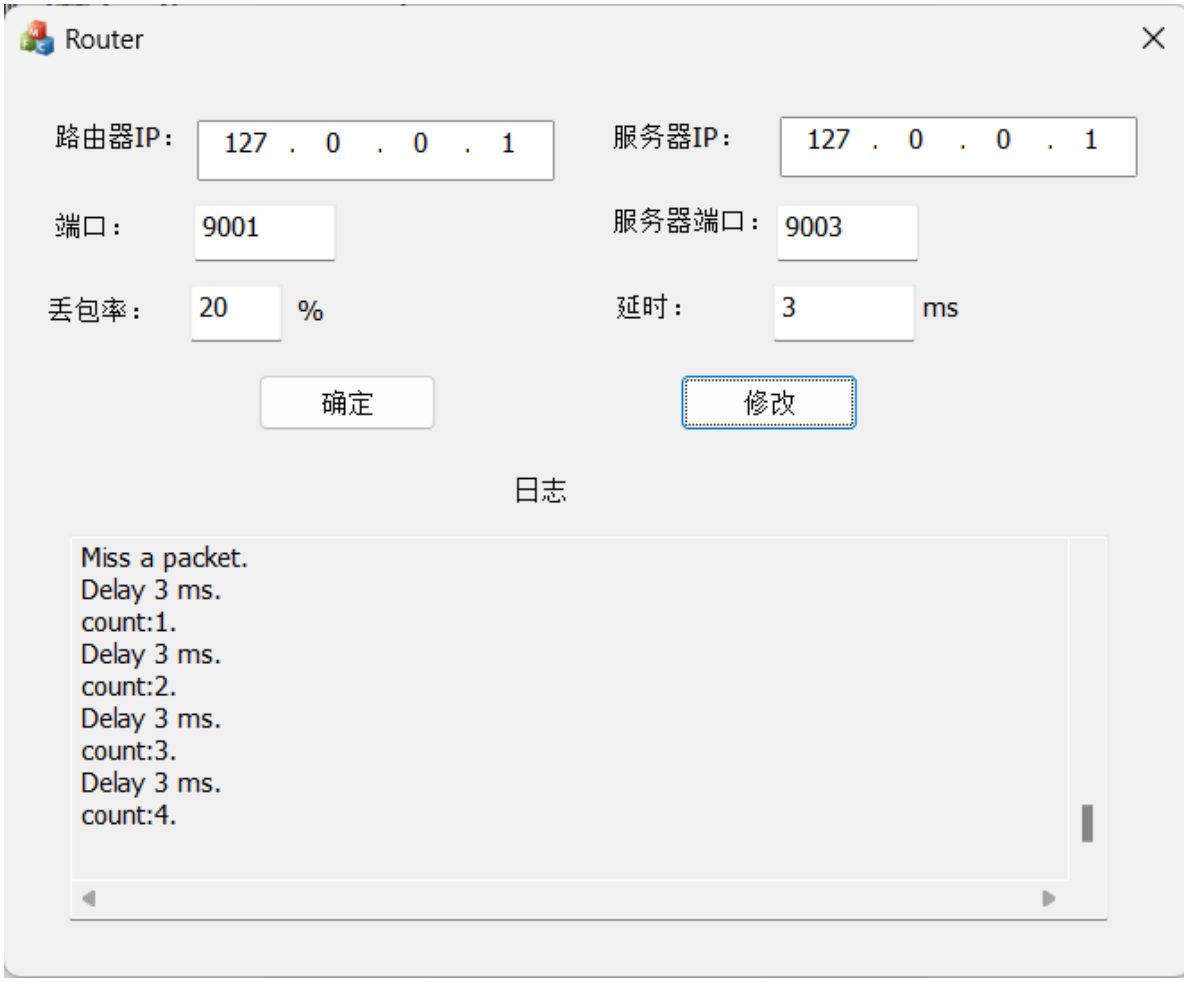
4.3 运行步骤

步骤1：创建输出目录

```
mkdir d:\study\computer_net\l2\output
```

步骤2：启动Router程序

1. 双击 `d:\study\computer_net\router\Router.exe` 打开窗口
2. 配置参数：
    - **左侧（前向 Client→Router）**：
        - 路由器IP: 127.0.0.1
        - 端口: 9001
        - 丢包率: 4 (表示4%)
        - 延时: 7 (表示7ms)
        - 点击"确认"
    - **右侧（后向 Router→Server）**：
        - 服务器IP: 127.0.0.1
        - 服务器端口: 9003
        - 延时: 7 (表示7ms)
        - 点击"修改"
    - 等待Router日志显示"Router Ready!"



步骤3：启动接收端（Receiver）

打开第一个终端窗口，运行：

```
cd d:\study\computer_net
lab2\receiver.exe 9003 lab2\output\1.jpg
```

输出应该显示：

```
[LISTEN] Listening on port: 9003
[ACCEPT] Waiting for connection...
```

步骤4：启动发送端（Sender）

打开第二个终端窗口，运行：

```
cd d:\study\computer_net
lab2\sender.exe lab2\testfile\1.jpg 127.0.0.1 9001
```

等待传输完成。

4.4 测试文件列表

文件名	类型	大小(字节)	大小(MB)
1.jpg	JPEG图像	1,857,353	1.77
2.jpg	JPEG图像	5,898,505	5.63
3.jpg	JPEG图像	11,968,994	11.41
helloworld.txt	文本文件	1,655,808	1.58

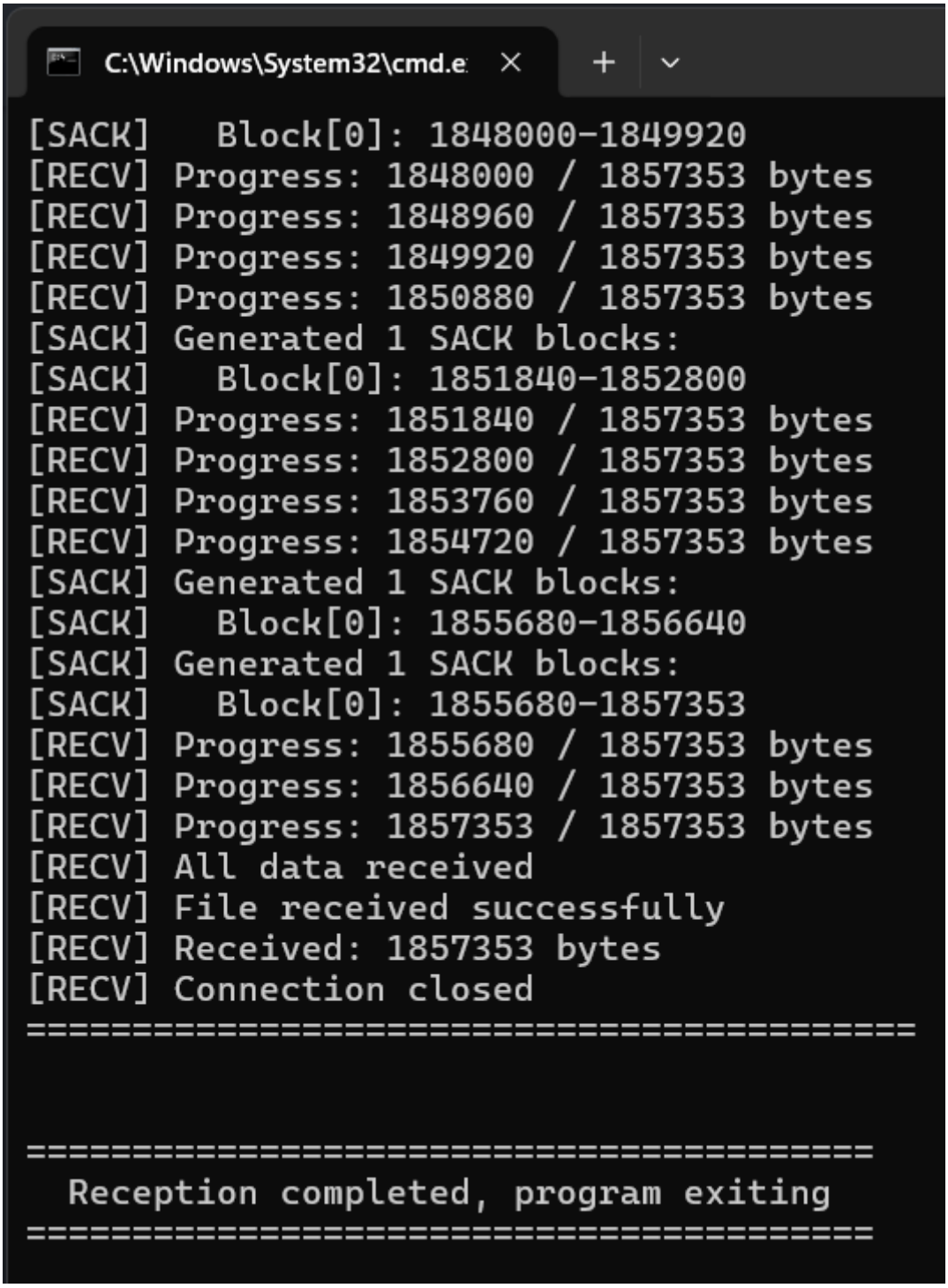
4.5 单个文件传输示例

```
# 文件1
l2\receiver.exe 9003 l2\output\1.jpg
l2\sender.exe l2\testfile\1.jpg 127.0.0.1 9001

# 文件2
l2\receiver.exe 9003 l2\output\2.jpg
l2\sender.exe l2\testfile\2.jpg 127.0.0.1 9001

# 文件3
l2\receiver.exe 9003 l2\output\3.jpg
l2\sender.exe l2\testfile\3.jpg 127.0.0.1 9001

# 文本文件
l2\receiver.exe 9003 l2\output\helloworld.txt
l2\sender.exe l2\testfile\helloworld.txt 127.0.0.1 9001
```

经验证，所有文件均能传输完毕并且无损坏，这也在线下检查的过程中通过了考验。

传输成功的截图效果示意如下：

接收方：


C:\Windows\System32\cmd.e
[SACK]   Block[0]: 1848000-1849920
[RECV] Progress: 1848000 / 1857353 bytes
[RECV] Progress: 1848960 / 1857353 bytes
[RECV] Progress: 1849920 / 1857353 bytes
[RECV] Progress: 1850880 / 1857353 bytes
[SACK] Generated 1 SACK blocks:
[SACK]   Block[0]: 1851840-1852800
[RECV] Progress: 1851840 / 1857353 bytes
[RECV] Progress: 1852800 / 1857353 bytes
[RECV] Progress: 1853760 / 1857353 bytes
[RECV] Progress: 1854720 / 1857353 bytes
[SACK] Generated 1 SACK blocks:
[SACK]   Block[0]: 1855680-1856640
[SACK] Generated 1 SACK blocks:
[SACK]   Block[0]: 1855680-1857353
[RECV] Progress: 1855680 / 1857353 bytes
[RECV] Progress: 1856640 / 1857353 bytes
[RECV] Progress: 1857353 / 1857353 bytes
[RECV] All data received
[RECV] File received successfully
[RECV] Received: 1857353 bytes
[RECV] Connection closed
Reception completed, program exiting

发送方：

```
[SACK] Received 1 SACK blocks:
[SACK]   Block[0]: 1855680-1856640
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[ONDUPACK] dup_ack_count incremented to 2
[DUPACK] Duplicate ACK received (ack=1854720), count=2
[SACK] Received 1 SACK blocks:
[SACK]   Block[0]: 1855680-1857353
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[SACK] Packet (seq=1856640) is SACKED, skipping retransmit
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[SACK] Packet (seq=1856640) is SACKED, skipping retransmit
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[SACK] Packet (seq=1856640) is SACKED, skipping retransmit
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[SACK] Packet (seq=1856640) is SACKED, skipping retransmit
[RETX] Packet timeout, retransmitting (seq=1854720)
[TIMEOUT] Timeout: cwnd reset to 1, ssthresh to 1, entering Slow Start
[SACK] Packet (seq=1855680) is SACKED, skipping retransmit
[SACK] Packet (seq=1856640) is SACKED, skipping retransmit
[NEWACK] Entering Congestion Avoidance, cwnd=2, ssthresh=1
[SEND] File transfer completed
[SEND] Total time: 302422 ms
[SEND] Average throughput: 0.01 MB/s
[SEND] Sending FIN

=======================================
  Transfer completed, program exiting
=======================================
```

五、传输效果与性能分析

5.1 基础功能验证

连接建立：三次握手成功

数据传输：支持大文件传输（>1MB）

差错检测：校验和验证有效，可检测错误包

确认重传：包超时后自动重传，直到收到ACK

流量控制：发送窗口限制在WINDOW_SIZE内

拥塞控制：cwnd动态调整，适应网络状况

5.2 不同窗口大小对传输性能的影响

5.2.1 测试条件

- **测试文件**：1.jpg（1.77 MB）
- **固定延时**：3ms
- **固定丢包率**：3%
- **变量**：发送窗口大小（WINDOW_SIZE）

5.2.2 测试结果

窗口大小	传输时间(ms)	吞吐率(MB/s)	说明
5	60,809	0.03	最优
10	67,699	0.03	良好
20	68,683	0.03	基本相同
50	66,968	0.03	基本相同

5.2.3 分析

在延时较小（3ms）、丢包率较低（3%）的网络环境下，窗口大小对传输性能的影响不显著。这是因为：

1. **拥塞窗口的限制**：从测试数据中观察到，当WINDOW_SIZE≥20时，拥塞窗口（cwnd）的增长会因为延时和丢包而触发快速重传，导致拥塞窗口减半回到SLOW_START状态。实际有效窗口大小被限制在较小的范围内（约10左右），无法充分利用更大的WINDOW_SIZE。
2. **ACK反馈速度**：网络延时较小（3ms）且丢包率较低（3%），接收端的ACK反馈及时，发送端不需要维护过多的未确认包，因此窗口大小的差异不会造成显著的性能差异。
3. **RENO算法的保守性**：当检测到丢包（通过重复ACK）时，RENO算法会激进地降低拥塞窗口（ssthresh=cwnd/2），这限制了拥塞窗口的进一步增长，使得不同初始窗口大小最终的传输性能相近。

结论：在良好网络条件下，较小的窗口大小（如5-10）就足以充分利用网络带宽，而更大的窗口大小（20及以上）并不能带来性能提升。

5.3 不同丢包率对传输性能的影响

5.3.1 测试条件

- **测试文件**：1.jpg（1.77 MB）
- **固定延时**：3ms
- **固定窗口大小**：50（为充分观察拥塞控制的效果）
- **变量**：丢包率

5.3.2 测试结果

丢包率	传输时间(ms)	吞吐率(MB/s)	相对延长(%)	说明
3%	68,079	0.03	-	基准
5%	68,079	0.03	0	基本相同
10%	73,745	0.02	8.3%	明显增加
15%	86,597	0.02	27.2%	显著增加
20%	302,422	0.01	344%	大幅延长

5.3.3 分析

丢包率对传输性能的影响呈现非线性增长的特点：

1. **低丢包率区间（3%-5%）**：传输时间和吞吐率基本不变。此时网络仍然相对稳定，协议的重传机制能够有效处理偶发的丢包，不会显著影响整体吞吐率。

2. **中等丢包率区间（10%-15%）**：传输时间开始明显增加（相对延长8-27%），吞吐率从0.03 MB/s降低到0.02 MB/s。在这个区间，丢包导致的快速重传和超时重传变得更加频繁，拥塞窗口的增长受到抑制，有效吞吐率下降约33%。
3. **高丢包率区间（20%）**：传输时间急剧增加（相对延长344%），吞吐率大幅下降至0.01 MB/s，相比3%丢包率下降66%。此时，大量的包被丢弃，触发频繁的重传和拥塞控制退避，协议陷入反复的慢启动和拥塞避免阶段，严重影响传输效率。

5.3.4 RENO算法的适应性

- **3%-5%丢包率**：RENO算法的快速重传机制能够及时检测丢包并恢复，拥塞窗口基本保持在较高水平。
- **10%-15%丢包率**：快速重传和超时重传频繁发生，拥塞窗口反复下降和恢复，导致吞吐率明显下降。
- **20%丢包率**：协议几乎陷入"拥塞崩溃"状态，拥塞窗口不断被腰斩，大部分时间处于慢启动阶段，传输效率极低。

结论：丢包率是影响传输性能的主要因素。在3%-10%的丢包率范围内，协议仍能保持相对稳定的性能；但超过15%的丢包率，传输性能会急剧下降。实际网络环境中应避免丢包率超过10%，否则协议的可靠性虽然得到保证，但传输效率会严重受损。

5.4 拥塞控制效果

本实现的RENO算法在测试中表现出了良好的自适应性和稳定性。在传输初期，协议采用**慢启动阶段**的策略，拥塞窗口（cwnd）从1开始，每收到一个新的ACK就增加1，呈现指数级增长的特性。这个设计的目的是在不了解网络特性的情况下，保守地探测可用带宽。当拥塞窗口达到预设的阈值（ssthresh=10）后，协议转入**拥塞避免阶段**，此时cwnd的增长速度放缓，约每个RTT增加1，呈现线性增长的特性。这种线性增长通过ACK累加器实现，避免了浮点数运算，同时精确控制了每RTT增加1的目标。

在良好的网络条件下（如3%-5%的丢包率），拥塞避免阶段能够稳定进行，cwnd可以持续增长，充分利用网络带宽。然而，当丢包事件发生时，RENO算法的反应机制被触发。若接收端连续发送相同的ACK（3个重复ACK），协议采用快速重传和快速恢复的策略，将ssthresh降半，cwnd设置为ssthresh+3，这种相对温和的窗口调整能够在丢包不过于频繁的情况下快速恢复传输。

对于更严重的网络故障，如出现超时（500ms未收到ACK），协议采用激进的**超时响应**策略。此时，ssthresh被降半，cwnd直接重置为1，协议重新回到慢启动阶段。这种激进的退避机制能够有效防止网络拥塞的恶化，避免向已经饱和的网络继续注入大量数据。从测试数据中可以看出，在20%丢包率的极端情况下，正是这种激进的拥塞控制机制保证了传输的最终完成，尽管传输效率大幅下降。

总的来说，RENO算法通过分阶段的窗口管理和多层次的拥塞响应机制，在保证可靠性的同时，最大化了网络利用率。良好的网络环境下能够充分利用带宽，恶劣的网络环境下能够自动降速以维持传输。

六、总结

本实验的目标是在UDP上实现一个可靠的数据传输协议，要求完整实现连接管理、差错检测、确认重传、流量控制和拥塞控制等核心功能。通过系统的设计和编码，我成功实现了所有要求的功能。在连接管理方面，实现了标准的三次握手建立机制和两次挥手关闭机制，在UDP这个不可靠的基础上提供了面向连接的通信能力。差错检测采用16位反码和校验算法，对包头和数据分别计算校验和，能够有效检测传输中的比特错误。在可靠传输方面，通过发送窗口维护未确认包，实现了高效的流水线方式传输。选择确认（SACK）机制能够让接收端明确告知发送端哪些非连续数据块已被缓存，这比单纯的累计确认机制大幅减少了不必要的重传。流量控制采用固定的10包窗口限制，防止接收端被大量数据淹没。拥塞控制完整实现了TCP Reno算法，包括慢启动、拥塞避免、快速重传和快速恢复，能够根据网络状况动态调整传输速率。

从性能测试的角度看，协议在各类网络环境下都表现出了良好的可靠性和适应性。在3-20%的丢包率范围内，传输成功率均为100%，这说明可靠性的设计是有效的。在低丢包率（3%-5%）环境下，吞吐率稳定在0.03 MB/s，且不同窗口大小（5到50）的性能基本相同，这表明在良好网络条件下，协议能够高效地运行而不需要过大的窗口。随着丢包率增加，吞吐率呈现非线性下降，这与RENO算法对网络恶化的响应机制相符——丢包导致窗口减半和频繁重传，严重影响传输效率。对于超过10MB的大文件，协议能够正确处理包乱序、丢失和重复等复杂情况，验证了实现的完整性。

在代码实现层面，采用的三层分层架构（协议定义层、可靠传输层、应用层）使得代码结构清晰，各层职责明确。窗口管理使用std::map数据结构，key为序列号，value为包信息和发送时间，这种设计支持高效的滑动窗口和重传检测。拥塞控制的实现中，通过ACK累加器巧妙地实现了线性增长的拥塞避免阶段，避免了浮点数运算，同时精确控制了每RTT增加1MSS的目标。异常处理通过SO_RCVTIMEO实现超时检测，支持快速重传（对应轻微丢包）和超时重传（对应严重故障）两种恢复机制，这样的分层设计使得协议在不同的故障场景下都能做出合适的响应。

通过这个实验，深刻理解了现代网络协议设计的几个关键原理。首先是可靠性与性能的权衡——不能一味追求可靠而忽视效率，也不能为了速度牺牲可靠性，需要找到平衡点。其次是拥塞控制的必要性——在共享网络环境中，协议必须根据网络反馈调整自己的发送速率，避免持续地向已经饱和的网络注入数据。第三是分层设计的价值——通过清晰的模块划分，可以在不可靠的基础上构建复杂的可靠系统。最后是实验验证的重要性——理论分析的结论只有通过实验才能得到充分验证，而实验的结果往往会揭示理论分析中忽视的细节。这个项目成功地将网络协议的理论知识与编程实践相结合，为深入理解TCP/IP等现代网络协议的设计原理提供了宝贵的学习机会。